Appears in the 29th Annual International Symposium on Computer Architecture (ISCA-2002), May 2002

Efficient Dynamic Scheduling Through Tag Elimination

Dan Ernst and Todd Austin
Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{ernstd,austin}@eecs.umich.edu

Abstract

An increasingly large portion of scheduler latency is derived from the monolithic content addressable memory (CAM) arrays accessed during instruction wakeup. The performance of the scheduler can be improved by decreasing the number of tag comparisons necessary to schedule instructions. Using detailed simulation-based analyses, we find that most instructions enter the window with at least one of their input operands already available. By putting these instructions into specialized windows with fewer tag comparators, load capacitance on the scheduler critical path can be reduced, with only very small effects on program throughput. For instructions with multiple unavailable operands, we introduce a last-tag speculation mechanism that eliminates all remaining tag comparators except those for the last arriving input operand. By combining these two tag-reduction schemes, we are able to construct dynamic schedulers with approximately one quarter of the tag comparators found in conventional designs. Conservative circuit-level timing analyses indicate that the optimized designs are 20-45% faster and require 10-20% less power, depending on instruction window size.

1. Introduction

In an effort to secure higher levels of system performance, microprocessor designs often employ dynamic scheduling as a technique to extract instruction level parallelism (ILP) from serial instruction streams. Conventional dynamic scheduler designs house a "window" of candidate instructions from which ready instructions are sent to functional units in an out-of-order data flow fashion. The instruction window is implemented using large monolithic content addressable memories (CAMs) that track instructions and their input dependencies.

While more ILP can be extracted with a larger instruction window (and accordingly larger CAM structure), this increased parallelism will come at the expense of a slower scheduler clock speed. Recent circuit-level studies of dynamic scheduler logic have shown that the scheduler CAM logic will dominate the latency for the structure [12], and as such, window sizes cannot be increased without commensurate increases in scheduler operation latency. More recent studies [1] also suggest that increasing wire latencies due to parasitic capacitance effects may make these trade-offs even more acute, with future designs seeing little benefit from smaller technologies. The optimal design is dependent on both the degree to which ILP can be harvested from the workload and the circuit characteristics of the technology used to implement the scheduler.

In addition to performance, power dissipation has become an increasing concern in the design of high-performance microprocessors. Increasing clock speeds and diminishing voltage margins have combined to produce designs that are increasingly difficult to cool. Additionally, embedded processors are more sensitive to energy usage as these designs are often powered by batteries. Empirical [9,7] and analytical [2,5] studies have shown that the scheduler logic consumes a large portion of a microprocessor's power and energy budgets, making the scheduler a prime target for power optimizations. For example, the scheduler components of the PentiumPro microarchitcture consume 16% of total chip power. A similar study for Compaq's Alpha 21264 microprocessor found that 18% of total chip power was consumed by the scheduler. Increasing window sizes and parasitic capacitances will continue to shift more of the power budget towards the scheduler.

Our techniques draw from the observation that most scheduler tag comparisons are superfluous to the correct operation of the instruction scheduler. Analyses reveal that most instructions placed into the instruction window do not require two source tag comparators because one or more operands are ready, or the operation doesn't require two register operands.

In this paper, we propose two scheduler tag reduction techniques that work together to improve the performance of dynamic scheduling while at the same time reducing power requirements. First, we propose a reduced-tag scheduler design that assigns instructions to reservation stations with two, one, or zero tag comparators, depending on the number of operands in flight. To reduce tag comparison requirements for instructions with multiple operands in flight, we introduce a last tag speculation technique. This approach predicts which input operand of an instruction will arrive last, and then schedules the execution of that instruction based solely on the arrival of this operand. Since the earlier arriving tags do not precipitate execution of the instruction, the scheduler can safely elim-

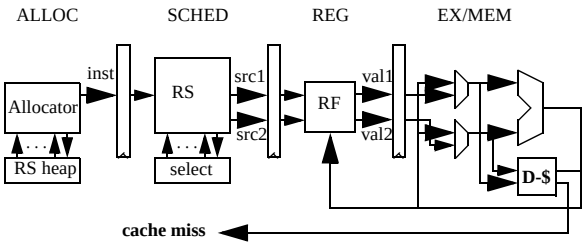

Figure 1: Conventional Dynamic Scheduler Pipeline

inate the comparator logic for all but the last arriving operand. A low cost and low latency misprediction recovery technique is presented.

The remainder of this paper is organized as follows. Section 2 gives background details into the design of a conventional high-performance scheduler, including the critical paths of the design and further motivation as to why removing comparators could improve its power and speed characteristics. Section 3 introduces our reduced-tag scheduler designs. Section 4 details our experimental evaluation of these new designs. Detailed cycle-accurate simulations and circuit-level timing and power analyses are combined to fully explore the benefits and costs of each approach. Section 5 details related prior work, and suggests how our approach could be combined with much of the previous work for further improvements in scheduler design. Section 6 gives conclusions and suggests future directions in the pursuit of high-performance schedulers. Appendix A includes a detailed description of our scheduler circuit design and analysis.

2. High-Performance Dynamic Scheduling

2.1. Scheduler Pipeline Overview

Figure 1 details the pipeline stages used to implement a high performance dynamic scheduler. The first stage, the allocator (ALLOC), is responsible for reserving all resources necessary to house an instruction in the processor instruction window. These resources include reservation stations, re-order buffer entries, and physical registers. Physical registers and re-order buffer entries typically use a FIFO allocation strategy in which the resources are allocated from a circular hardware queue.[1] This approach works well because resources are allocated in program order in ALLOC and held until instruction commit time, where the resources are released in program order. Reservation stations, while allocated in program order, may be released as soon as an instruction has begun execution (or as soon as an instruction has begun execution for the last time in a pipeline with replay/re-execution support). As such, a heap-style allocation strategy will result in more efficient use of reservation station resources than can be attained using a FIFO allocation policy.

1. While it is conceivable that physical registers could delay allocation until writeback, *i.e.,* when the resource is needed to store the result, most designs avoid any type of out-of-order allocation because it introduces many deadlock scenarios. A good treatment on out-of-order register allocation and its potential hazards can be found in [6].

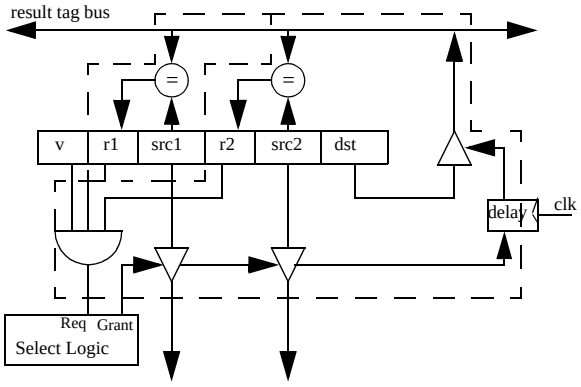

Figure 2: Reservation Station Datapaths and Control. Critical path shown with dashed line.

The scheduler stage (SCHED) houses instructions in reservation stations until they are ready to execute. Reservation stations track the availability of instruction source operands. When all input operands are available, a request is made to the select logic for execution. The selection logic chooses the next instructions to execute from all ready instructions, based on the scheduler policy. The selected instructions receive a grant signal from the selection logic, at which point they will be sent forward to later stages in the pipeline.

Once granted execution, an instruction's source register tags are used to access the register file in the register read (REG) stage of the pipeline. In the following stage, operand values read from the register file are forwarded to the appropriate functional unit in the execute stage (EX/MEM) of the pipeline. If a dependent operation immediately follows an instruction, it will read a stale value from the physical register file. A bypass MUX is provided in the EX/MEM stage to will select between the incoming register operand, or a more recent value on the bypass bus. Dependent instructions that execute in subsequent cycles must communicate via the bypass bus. All other instructions communicate by way of the physical register file.

2.2. Reservation Stations

Figure 2 illustrates the datapaths and control logic contained in each reservation station. Each new instruction is placed into a reservation station by the allocator. If an instruction's input operand has already been computed (or if the operand is not used by the instruction), the ready bit for that operand is set as valid. If the operand has not yet been computed, a unique tag for the value is placed into the corresponding source operand tag field, either *src1* or *src2* depending on which instruction operand is being processed. Since all input operands are renamed to physical storage, the physical register index suffices as a unique *tag* for each value in flight within the instruction window. Unlike most textbook descriptions of Tomasulo's algorithm [14], most modern processors, such as the Alpha 21264 [8] and Pentium 4, use value-less reservation stations. Instead of storing instruction operand values and

opcodes in the CAM structure and making longer tag busses, these designs keep this data in the REG stage where it can be accessed on the way to execution.

When instructions are nearing the completion of their execution, they broadcast their result tag onto the result tag bus. Reservation stations snoop the result tag bus, waiting for a tag to appear that matches either of their source operand tags. If a match is found, the ready bit of the matching operand tag is set. When a valid reservation station has both operands marked ready, a request for execution is sent to the selection logic. The selection logic grants the execution request if the appropriate functional unit is available and the requesting instruction has the highest priority among instructions that are ready to execute. Policies for determining the highest priority instruction vary. Some proposed approaches include random [12], oldest-first [12], and highest-latency first [18]. However, more capable schedulers require more complex logic and thus run more slowly.

The selection logic sends an instruction to execution by driving its grant signal. The input operand tags are driven onto an output bus where they are latched for use by the REG stage in the following cycle. In addition, the grant signal is latched at the reservation station. In the following cycle, the instruction will drive its result tag on to the result tag bus. If the execution pipeline supports multi-cycle operations, the result tag broadcast must be delayed until the instruction result is produced. This can be implemented by inserting a small delay element into the grant latch, such as a small counter. If the execution time for an instruction is non-deterministic, such as for a memory operation, the scheduler can optimistically predict that the latency will be the most common case; *e.g.*, it predicts that all loads will hit in the data cache. If an instruction's latency is mispredicted, dependent instructions were scheduled too soon and must be rescheduled to execute after the operation completes. This rescheduling is sometimes called a scheduler replay [20].

The reservation station wakeup and select logic forms the control critical path in the dynamically scheduled pipeline [12]. This logic forms a critical speed path in most aggressive designs because it limits the rate at which instructions can begin execution. As shown by the dashed lines in Figure 2, the scheduler critical path includes the result tag driver, the result tag bus interconnect, the reservation station comparators, the selection logic, and the grant signal interconnect. It is possible that the operand tag output busses (src1 and src2) are on the critical path of the control loop, however, in aggressive designs this output can be pipelined or wave-pipelined [11] into subsequent scheduler cycles because the output bus value is not required to initiate the next scheduler loop iteration. As noted by Palacharla et al [12], the CAM structure formed by the result tag drivers, result tag bus, and comparators constitute the major portion of the control circuit latency, especially for large windows with many reservation stations.

3. Reduced-Tag Scheduler Designs

In this section, we present two reduced-tag scheduler designs. The first optimization, called window specialization, leverages the observation that many instruction input operands are available or unneeded when instructions are placed into a reservation station. As a result, these instructions can be scheduled with reservation stations containing fewer tag comparators. The second optimization uses a last-tag predictor to identify the operand of an instruction that will arrive last (and thus allow the instruction to commence execution). A design that can effectively make this prediction can eliminate the tag comparators of all other operands without impacting the dynamic instruction schedule.

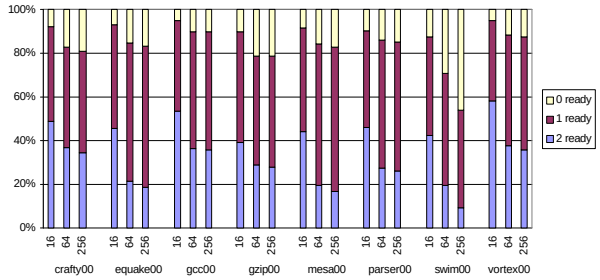

Figure 3: Runtime Distribution of Ready Input Operands for Varying Window Sizes

3.1. Specialized Windows

When an instruction enters the instruction window, its input operand tag fields are loaded with the index of the physical register that will eventually hold the operand value. It may be the case that some of the operands will be ready at that time, either because the operand was computed in an earlier cycle or the operand is not required by the operation (e.g., one of the operands is an immediate value). Since these input operands are already available, their reservation station entries do not require tag comparators.

To quantify the degree to which tag comparators are not required by reservation stations, a typical 4-wide superscalar processor was simulated using the SimpleScalar toolset [3] with instruction window sizes of 16, 64, and 256 and an load/store queue size that was half the size of the instruction window. When instructions entered the scheduler, the number of ready input operands was counted. The results in Figure 3 show the dynamic distribution of the number of ready operands for all instructions. Results are shown for eight of the SPEC2000 benchmarks [17]. (More details on our experimental framework and baseline microarchitecture model can be found in Section 4.1.)

Clearly, a significant portion of all operands are marked ready when they enter reservation stations, for all instruction window sizes. Only about 10-20% of all dynamic instructions require a reservation station with two tag comparators, while the remaining instructions require either one or zero comparators. Much of this effect comes from the fact that very few instructions (20-35%) actually have two architectural operands (many are loads/stores or use immediates). As expected, larger window sizes result in fewer ready operands, because larger windows permit the front-end to get further ahead of instruction execution. Nonetheless, a window size of 256 instructions has a significant portion of instructions that do not require more than one tag comparator. In general, programs with poor

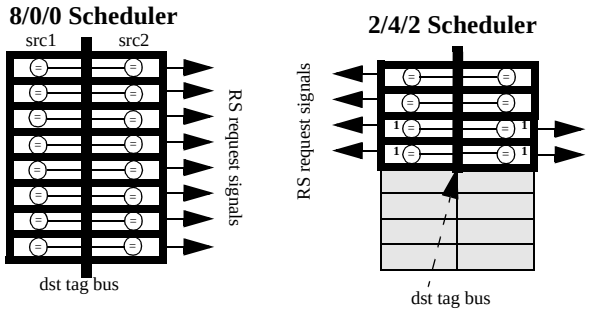

Figure 4: Conventional and Reduced-Tag Reservation Stations. The circles represent tag comparators. The bold tag entries include a comparator, the shaded tag entries are not necessary and so do not include comparators. One-tag entries are denoted with a "1".

branch prediction such as *GCC* and *Vortex* were less affected by the larger windows sizes, because branch mispredictions limit the degree to which the front-end can get ahead of instruction execution. In contrast, *SWIM* has nearly perfect branch predictor accuracy, which results in more slip between fetch and execute for large window sizes. Still, even in this extreme example, more than half of the instructions in a 256-entry instruction window require less than two tag comparators. Similar observations were made by Folegnani and Gonzalez [5]; they used this property to design low-power tag comparators.

It is possible to take advantage of ready input operands if the scheduler contains reservation stations with fewer than two tag comparators. Figure 4 illustrates a reservation station design that contains entries with two, one, and zero tag comparators. The design on the left side of the figure is a conventional scheduler configuration, where each reservation station contains two tag comparators. The optimized design, shown on the right side of the figure, eliminates tag comparators from some of the reservations stations. We label the configurations "x/y/z", where x, y, and z indicate the number of two, one and zero tag stations, respectively.

When the allocator encounters an instruction with one or more unavailable operands, the allocator will assign the instruction to a reservation station with a matching number of tag comparators. If both operands are ready, we can place the instruction into a reservation station without tag comparators that immediately requests execution. If there isn't an available reservation station with the same number of tag comparators, the allocator will assign the instruction to a reservation station with more tag comparators. For example, instructions waiting for one operand can be assigned to reservation stations with one or two tag comparators. Finally, if a reservation station with a sufficient number of tag comparators is not available, the allocator will stall the front-end pipeline until one is available.

This reduced-tag scheduler design has two primary advantages over a conventional design. First, the destination tag bus, which drives a physical register destination tag to all source tag comparators, need only run to the reservation stations with tag comparators. Since result tag drive latency is on the critical path of the control scheduler

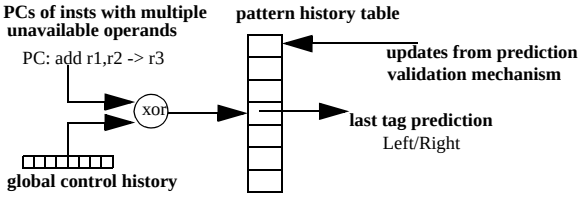

Figure 5: GSHARE-Style Last Tag Predictor

loop, the latency of this critical path will be reduced in proportion to the number of zero-tag reservation stations. The second advantage is that comparator circuits can be eliminated from the instruction window. With fewer comparators, load capacitance on the result tag bus is reduced, resulting in faster tag drive and lower power requirements. The downside of the reduced tag design is that additional allocator stalls may be introduced when there are insufficient reservation stations of a required class, potentially reducing extracted ILP and program performance.

3.2. Last Tag Speculation

While many instructions enter the instruction window with multiple unavailable operands, it is still possible to eliminate all but one of the tag comparators for these instructions. Since the arrival of the all but the last input operand tags will not initiate an execution request, these tag comparators can be safely removed. When the *last* input operand tag arrives, this sole event can be used to initiate instruction execution. We employ a last-tag predictor to predict the last arriving operand. As long as the last tag predictor is correct, the schedule will proceed as in the non-speculative case. A modified reservation station with one tag comparator monitors the arrival of the predicted last input operand.

We experimented with a number of last-tag predictors, including a predictor that predicts the last operand to arrive will be the same as in the previous execution, a bimodal-type last-tag predictor (similar to the grandparent predictor employed by Stark et al [19]), and a GSHARE-style last-tag predictor. The accuracy of these predictors was nearly identical to similarly configured branch predictors. For the sake of brevity we only present the GSHARE-style last-tag predictor as it consistently performed the best with only marginal additional cost over simpler predictors.

Figure 5 illustrates the GSHARE-style last-tag predictor. The predictor is indexed with the PC of an instruction (with multiple unavailable operands) hashed with global control history [10]. The control history is XOR'ed onto the least significant bits of the instruction PC and that result is used as an index into a table of two-bit saturating counters. The value of the upper counter bit indicates the prediction: one indicates the left operand will arrive last, zero indicates the right operand will arrive last. The predictors are updated when last tag predictions are validated. Figure 6 shows the prediction accuracy (for instructions with two operands in flight) for various sizes of the GSHARE-style last-tag predictors with 8 bits of control history and an instruction window size of 64. Most programs have good predictor performance for sizes larger than 1024 entries. We also implemented a confidence estimation technique for the predictor. The approach forced

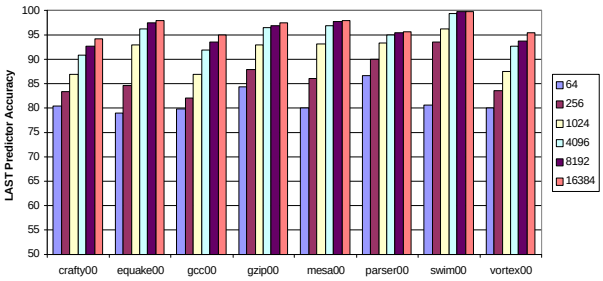

Figure 6: Accuracy of GSHARE-Style Last Tag Predictor for Various Predictor Sizes

hard to predict instructions to use two-tag entries, however, the results showed few gains.

As shown in the pipeline of Figure 7, the allocator accesses the last-tag predictor for instructions with multiple unavailable operands and inserts the instruction into a reservation station with a single tag comparator. The predictor will indicate whether the left or right operand for the instruction will complete last. If the last tag prediction is correct, the instruction will wake up at the exact same time it would have in a window without speculation. In the event that the prediction is incorrect, the instruction will wake up before all of its input operands are ready, and a mispeculation recovery sequence will have to be initiated.

Figure 8 illustrates the datapaths and control logic for a reservation station supporting last-tag speculation. The input operand tags are loaded into the reservation station with the tag predicted to arrive last placed under the comparator (*srcL*). The other input operand tag (*srcF*) and the result tag are also loaded into the reservation station. The reservation station operates in a manner similar to a conventional design. Instructions request execution once the predicted-last tag is matched on the result tag bus. When an instruction is granted permission to execute, the source operand register tags are driven out to the register stage of the pipeline. This drive operation requires a pair of muxes to sort the source operands into the original (left, right) instruction order, which is the format used by the register file and later functional units. In addition, the tag predicted to arrive first is forwarded to the register read stage (REG), where it is used to check the correctness of the last-tag prediction.

The last-tag prediction must be validated to ensure that the instruction does not commence execution before all of its operands are available. The prediction is valid if the operand predicted to arrive first (*srcF*) is available when the instruction enters the register read stage (REG) of the pipeline. In parallel with the register file access, the *srcF* tag is used to probe a small register scoreboard (*RDY*). The scoreboard contains one bit per physical register; bits are set if the register value is valid in the physical register file. This scoreboard is already available in the ALLOC stage of the pipeline, where it is used to determine if the valid bit should be set when operand tags are written into reservation stations. A number of ports equal to the issue width added to this scoreboard will suffice for validating last-tag predictions. Alternatively, an additional scoreboard could be maintained specifically for last-tag prediction validation.

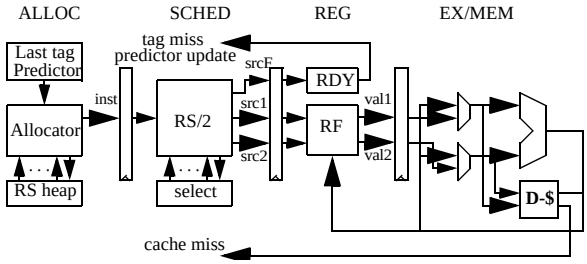

Figure 7: Scheduler Pipeline with Last Tag Speculation

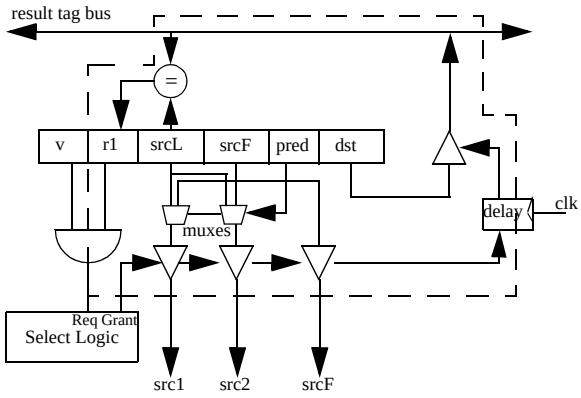

Figure 8: Reduced-Tag Reservation Station with Last Tag Speculation

If the prediction is found to be correct, instructions may continue through the scheduler pipeline as the scheduler has made the correct scheduling decision. If the prediction is incorrect, the scheduler pipeline must be flushed and restarted, in a fashion identical to latency mispredictions. Unlike latency mispredictions, which are detected in MEM with a three cycle penalty, last-tag misspeculations can be detected before EX, and thus only cause a one cycle bubble in the scheduler pipeline.

The primary advantage of the last tag scheduler is that more than half of the comparator load on the result tag bus is eliminated, which can result in reduced scheduling latency and significant power reductions for large instruction windows. The drawback of this approach is, of course, the performance impacts that result when a last-tag prediction is incorrect. Fortunately, the accuracy of the last tag predictor, combined with the small penalty for mispredicting should make this approach an effective technique for improving scheduler speed and energy consumption.

4. Experimental Evaluation

4.1. Methodology

The architectural simulators used in this study are derived from the SimpleScalar/Alpha version 3.0 tool set [3], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 4-way dynamically scheduled microprocessor with two levels of instruction and data

Table 1: Benchmarks and Baseline Statistics

Bench mark	Baseline IPC	Baseline IPns	Bench mark	Baseline IPC	Baseline IPns
crafty	2.018	4.330	mesa	2.695	5.783
equake	2.622	5.626	parser	1.736	3.725
gcc	1.675	3.593	swim	2.719	5.835
gzip	2.186	4.692	vortex	1.961	4.208

cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction.

To perform our evaluation, we collected results from eight of the SPEC2000 benchmarks [17]. There are five integer programs and three floating point programs. All SPEC programs were compiled for a Compaq Alpha AXP-21264 processor using the Compaq C and Fortran compilers under the OSF/1 V4.0 operating system using full compiler optimization (-O4). The simulations were run for at least 250 million instructions using the SPEC reference inputs and all simulations were fast-forwarded through the first 100 million instructions to warm up the caches and predictors.

To get a full understanding of the effects of our optimizations, the circuit characteristics of scheduler structures must be examined. The circuit delays and power consumption statistics for scheduler windows used in this study were derived from an updated version of the SPICE models used in the work by Palacharla, Jouppi, and Smith [13]. All timing results are for the TSMC 0.18 μm process; a more detailed description of our circuit models can be found in Appendix A. In addition, we estimated the power consumed by the last tag prediction array using CACTI II [15]. Table 1 shows the benchmarks, their instructions per cycle (IPC) on the baseline microarchitectural model, and their maximum baseline scheduler performance in instructions per ns (IPns).

Our baseline simulation configuration models a current generation out-of-order processor microarchitecture. It can fetch and issue up to 4 instructions per cycle and it has a 64 entry dynamic scheduler window with a 32 entry load/store buffer. There is an 3 cycle minimum branch misprediction penalty. The processor has 4 integer ALU units, 2-load/store units, 4-FP adders, 4-integer MULT/DIV units, and 4-FP MULT/DIV units. The latencies vary depending on the operation, but all functional units, with the exception of the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle.

The memory system consists of 32k 4-way set-associative L1 instruction and data caches. The data cache is dual-ported and pipelined to allow up to two new requests each cycle. There is also a 256k 4-way set-associative L2 cache with a 6 cycle hit latency. If there is a second-level cache miss it takes a total of 36 cycles to make the round trip access to main memory.

The model uses a GSHARE branch predictor with an 8-bit global history and an 8k entry BTB. The instruction fetch stage of the model has a 32 entry instruction queue and operates at twice the frequency as the rest of the processor. While perhaps not realistic, this assures that the IPC results seen when changing scheduler configurations are not attenuated by bottlenecks in the front end. Since improvement in scheduler performance would have to be accompanied by commensurate improvement in fetch bandwidth, we feel that this configuration will accurately portray the benefits of our scheduler optimizations.

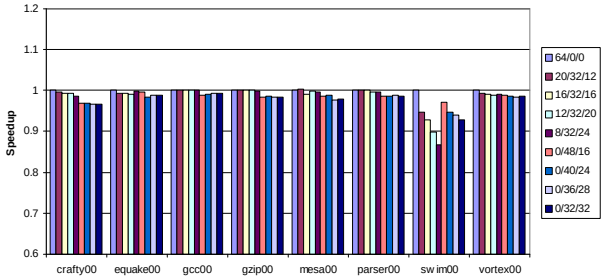

Figure 9: Normalized Instructions Per Cycle for Varying Configurations

The dynamic scheduler distributed with SimpleScalar is overly simplistic compared to modern schedulers. A redesigned scheduler was added to sim-outorder to more accurately reflect the design presented in Section 2. Our new scheduler includes support for scheduler replay, more efficient scheduler resource management, decoupled ROB and RS resources, and support for our optimizations detailed in Section 3.

The last-tag predictor configuration simulated is a GSHARE-style predictor with an 8-bit global history and a 8192 entry pattern history table. The global history is updated when branch instructions complete in the same way that the branch predictor is updated. A last-tag mispredict causes a 1 cycle bubble in the scheduler pipeline.

4.2. Performance of Reduced-Tag Schedulers

When reduced-tag reservation stations are introduced, instructions have a new constraint on entering the instruction window. Not only must there be an empty reservation station, but the station must also have at least one tag comparator for each of the instruction's unavailable input operands. If the demand for any particular class of reservation stations is high, the reduced-tag designs may experience extra instruction stalls as the allocator waits for reservation stations to be freed.

These extra stalls reduce the effective number of reservation stations from which the scheduler can choose instructions to execute. The result, as shown in Figure 9, is a small IPC change for most configurations and benchmarks. We label the configurations "x/y/z", where x, y, and z indicate the number of two, one and zero tag stations, respectively. Only the configurations without two-tag stations employ last-tag speculation. The effects of the stalls show up most prominently in *SWIM*. This benchmark makes extremely efficient use of the machine because it has excellent branch predictor performance, very few stalls, and many tightly coupled dependent instructions. Consequently, many instructions require the full two tag comparators. The configurations using last-tag speculation perform very well, with slightly lower

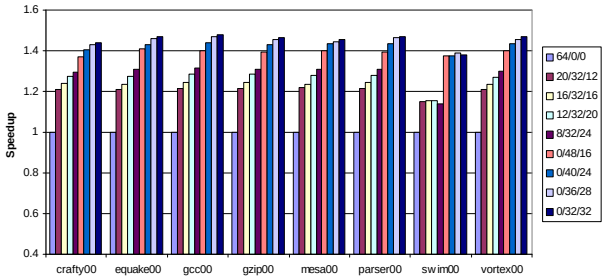

Figure 10: Normalized Instructions Per ns for Varying Configurations

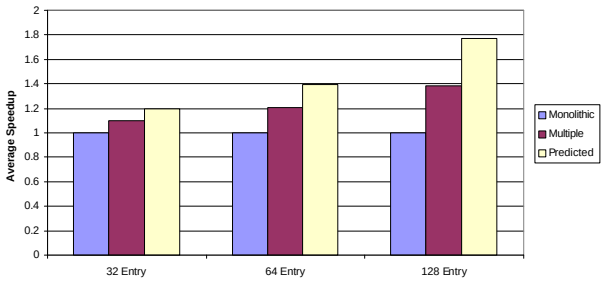

Figure 11: Impact of Tag-Reduction for Varying Window Sizes

IPC's seen in benchmarks with poor branch predictor accuracy, such as *crafty* and *GCC*. In these programs, complex program control causes the register dependencies between instructions to change rapidly, making it more difficult to predict which operand will arrive last. The configurations without last-tag speculation slightly outperformed the configurations with speculation. Overall, the performance impacts amounted to only 1-3%.

The main benefit of removing tags from the scheduler critical path is the reduction in the load capacitance during instruction wakeup. Lower load capacitance allows for more aggressive clocking of scheduler circuitry. Based on our model of the wakeup and select circuitry, the specialized windows should allow for 25-45% faster clock rates, depending on configuration. Figure 10 shows the total performance (measured in instructions per ns and assuming no other critical path bottlenecks in the system) of each benchmark. With the exception of *SWIM*, the rate at which the scheduler can send instructions to execute measures between 20-45% higher, again depending on configuration[1].

To more accurately gauge the impact on modern superpipelined processors, we also simulated the scheduler with last-tag misprediction latencies of 2 and 4 cycles. In these cases, the IPC of the configurations using the predictor

1. For the purposes of our study, we assume that the scheduler critical path is the limiting factor in determining the clock speed of the entire processor. This may or may not be true for actual full implementations. Alternatively, if another pipeline stage limits the clock speed gains that can be achieved, the performance headroom afforded by tag elimination can instead be used to make the scheduling window larger to provide more parallelism.

Table 2: Energy and Power

Scheduler	Energy (nJ)	Power (W)	Scheduler	Energy (nJ)	Power (W)
64/0/0	0.468	1.550	0/48/16	0.255	1.548
20/32/12	0.314	1.435	0/40/24	0.222	1.416
16/32/16	0.289	1.375	0/36/28	0.207	1.372
12/32/20	0.263	1.322	0/32/32	0.191	1.281
8/32/24	0.239	1.250			

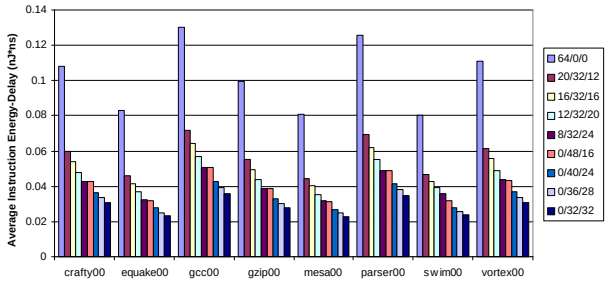

Figure 12: Energy-Delay Product for Varying Configurations

were reduced an average of 0.7% and 2.4%, respectively. In every case, there was still a substantial speedup in instructions per ns.

4.3. Impact of Window Size

Figure 11 shows that reduced-tag scheduler optimizations continue to pay dividends for differing window sizes. The results given are the averages across all benchmarks for monolithic, multiple (5:8:3 ratio), and predicted (0:3:1 ratio) style windows. The gains become more prominent as total window size grows. The larger windows have fewer allocator stalls due to more reservation station resources. The large windows also bear more of the scheduler latency in result tag broadcasts (as opposed to the select logic), as a result, they show a larger percentage gain when tag comparators are eliminated. For a window with 128 entries, the optimized schedulers were 35-75% faster.

4.4. Energy and Power Characteristics

Often it is the case that to reduce power consumption, design changes must be made at the cost of lower performance. In our reduced-tag scheduler designs, lower load capacitance on the result tag bus provides both performance and power benefits. Table 2 shows the energy consumption for each configuration. The optimized designs use 30-60% less energy than the standard monolithic scheduler. Table 2 also shows the power used by each configuration if it were to be run at its maximum possible clock speed. The power reductions are not as pronounced as the energy improvements because the optimized designs run at a faster clock rate.

The power usage of the last-tag predictor was also calculated. It was found to consume less than 10% of the power used by the scheduler in all cases.

One way to quantify an architecture's ability to balance both power and performance is through the use of the energy-delay product [7]. This metric is the product of program run-time and total energy consumed to run the program. Figure 12 shows that the energy-delay product of the optimized scheduler is 50-75% lower than the baseline configurations. The 0/32/32 speculative configuration had the best return, with a 65-75% lower energy-delay across all experiments, including *SWIM*, which had the largest IPC impacts. The optimized designs show large gains because eliminating tag comparators and tag bus wiring lowers the result tag bus capacitance, which both reduces energy consumption and allows for higher clock speeds. The energy-delay products for 128 entry windows also showed a 70% gain for the optimized configurations, suggesting that these benefits continue with larger window sizes.

5. Related Work

There have been several other efforts to reduce the complexity of dynamic schedulers, many of which can be used in combination with tag elimination.

Some current designs bank their selection logic by having separate groups of reservation stations for each group of functional units. Each of these station groups has its own, smaller, selection network. While result tag broadcasts still need to be sent to all of the reservation stations, the latency for selecting instructions for execution can be reduced. As a consequence of this optimization, the latency of the wakeup path makes up a higher percentage of the total scheduler delay. Because of this, as Figure 13 shows, schedulers with banked selection logic reap a larger benefit from using tag elimination techniques.

Palacharla, Jouppi, and Smith studied the effects of process scaling on microprocessor design and proposed a complexity-effective superscalar processor [12,13]. Their design uses a set of FIFO queues for dynamic scheduling to reduce complexity and allow for very aggressive clocking. Instructions can only be issued from the front of the queues; instructions are steered into them using dependence information. This approach attempts to maximize the number of ready instructions at the issue boundary.

Stark, Brown, and Patt have proposed two methods for pipelining wakeup and selection logic, allowing for a faster clock. For their first method [19], each reservation station entry carries its own input tags along with its parent instructions' input tags in order to allow back-to-back dependent instructions to execute consecutively. They also propose speculating on which parent instruction will finish last, reducing the number of "grandparent" tags that must be stored.

Their second method, select-free logic [22], enables pipelining by allowing all instructions that wakeup to broadcast back into the window the following cycle, even though some of them may not be selected for execution. Combining tag elimination with either of these schemes could be very lucrative because the latency of the wakeup stage is fully exposed.

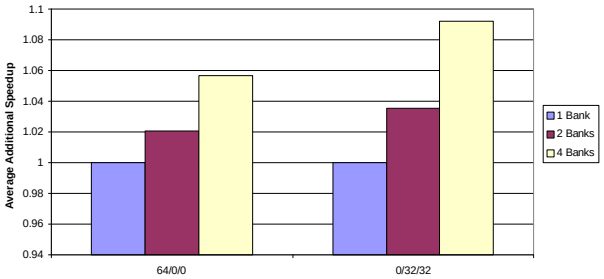

Figure 13: IPns Comparison of Banking Benefits

In their studies on energy-effective issue logic, Folegnani and Gonzalez [5] also made the observation that many comparators in the instruction window are unused and unnecessary. In their low-power scheduler design, tags that are marked ready do not precharge their match lines, resulting in lower comparator power consumption. This approach dynamically reduces the power consumption of the window, but it doesn't allow for a faster clock rate.

Gonzalez and Canal [23] also propose a way to reduce the overall complexity of scheduling logic by using N-Use issue scheme. Their optimization takes advantage of the observation that most instruction output values are ready only once.

Michaud and Seznec [21] proposed a method for reordering instructions as they enter the instruction window. By performing dependence analysis in a pre-schedule stage, they are able to place more usable instructions into the window, increasing its effective size.

Kucuk, *et al.* [24] propose an alternate comparator circuit to reduce energy dissipation in dynamic schedulers. Their optimization, like many of the others, could be used in combination with tag elimination for improved energy-efficiency.

6. Conclusions

The wakeup and select logic of dynamic schedulers has become one of the primary bottlenecks in high-performance microprocessor design. While architects have sought larger scheduling windows to allow for wider issue widths and higher IPCs, the circuit complexity of these devices forces any gains to be at the expense of clock speeds. Moreover, interconnect-intensive scheduling logic consumes a significant portion of processor design power budgets. Designers must be aware of all these factors when making scheduler design decisions because changes that improve one aspect of the design may adversely effect another.

We have introduced more efficient reduced-tag scheduler designs that improve both scheduler speed and power requirements. By employing more specialized window structures and last-tag speculation, a large percentage of tag comparisons were removed from the scheduler critical path. These optimizations reduced the load capacitance seen during tag broadcast while maintaining instruction throughputs that are close to those of inefficient monolithic scheduler designs. The optimized designs allow for more aggressive clocking and significantly reduce power consumption.

There are still many ideas to be explored in this area. There are potentially many improvements to be made in the last tag prediction mechanism. Specifically, in an effort to improve accuracy, factors that contribute to a change in issue order could be examined, such as branch mispredictions and instruction latency variations.

In addition, these techniques could be combined with many of the prior proposals detailed in the related work section to produce greater benefits. For example, tag elimination and last-tag prediction can be used to further reduce the complexity of banked-select scheduling logic or select-free logic. Similarly, the power-saving techniques introduced by Gonzalez could be combined with our scheduler for additional power savings or even for the purposes of allowing the logic to modify its own clock rate.

Acknowledgements

We would like to thank Chris Weaver and Matt Guthaus for their help with our CAD tools. We also thank all of the reviewers and our collegues for their insights and suggestions for strengthening our paper.

This work was supported by Contract No. 98-DT-660 to the Regents of the University of California from Microelectronic Advanced Research Corporation (MARCO) and by the National Science Foundation CADRE program, Grant No. EIA-9975286.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. “Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures”, *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[2] D. Brooks, V. Tiwari, and M. Martonosi. “Wattch: A framework for architectural-level power analysis and optimizations”, *In 27th Annual International Symposium on Computer Architecture*, June 2000.

[3] D. Burger and T. M. Austin. "The SimpleScalar tool set, version 2.0", Tech. Rep. CS-1342, University of Wisconsin-Madison, June 1997.

[4] Brian A Fields, Shai Rubin and Rastislav Bodik, “Focusing Processor Policies via Critical-Path Prediction”, *28th Annual International Symposium on Computer Architecture*, June 2001.

[5] Daniele Folegnani and Antonio Gonzalez, “Energy-Effective Issue Logic”, *In 28th Annual International Symposium on Computer Architecture*, June 2001.

[6] Antonio Gonzalez, Jose Gonzalez and Mateo Valero. “Virtual-Physical Registers”, *Proc. 4th Intl. Symp. High-Performance Computer Architecture* (HPCA-4), Feb 1998.

[7] Ricardo Gonzalez and Mark Horowitz. “Energy Dissipation in General Purpose Microprocessors”, *IEEE Journal of Solid-State Circuits*, 31(9):1277-1284, September 1996.

[8] Keller, J. 1996. “The 21264: a superscalar Alpha processor with out-of-order execution”, Presented at the 9th Annual Microprocessor Forum, San Jose, CA.

[9] S. Manne, D. Grunwald, A. Klauser, "Pipeline Gating: Speculation Control for Energy Reduction", *25th Annual International Symposium on Computer Architechture*, June 1998.

[10] Scott McFarling. “Combining branch predictors”, Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[11] Kevin J. Nowka, “High-Performance CMOS System Design using Wave Pipelining”, Stanford University Ph.D. Thesis, September 1995.

[12] Subbarao Palacharla, Norman P. Jouppi and J.E. Smith. "Complexity-Effective Superscalar Processors”, *In 24th Annual International Symposium on Computer Architecture*, May 1997.

[13] Subbarao Palacharla, Norman P. Jouppi and J.E. Smith. "Quantifying the Complexity of Superscalar Processors", Tech. Rep. CS-1328, University of Wisconsin-Madison, May 1997.

[14] Patterson, D. A. and Hennessy, J. L. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, second edition, 1996.

[15] Glenn Reinman and Norm Jouppi, “An Integrated Cache Timing and Power Model”, Compaq Technical Report, http://www.research.compaq.com/wrl/people/jouppi/cacti2.pdf.

[16] The MOSIS Service, http://www.mosis.com/Technical/Processes/proc-tsmc-cmos018.html

[17] SPEC System Performance Evaluation Committee, www.spec.org.

[18] S. T. Srinivasan, A. R. Lebeck. "Load Latency Tolerance in Dynamically Scheduled Processors", *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 148-159, 1998.

[19] J. Stark, M. Brown, and Y. Patt, “On Pipelining Dynamic Instruction Scheduling Logic”, *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, December 2000.

[20] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. “Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor”, In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191-202, May 22-24, 1996.

[21] P. Michaud, A. Seznec. “Data Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors”, *HPCA-7*. January 2001.

[22] M. Brown, J. Stark, and Y. Patt, “Select-Free Instruction Scheduling Logic”, To appear in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, December 2-5, 2001.

[23] Ramon Canal and Antonio Gonzalez, “Reducing the Complexity of the Scheduling Logic”, *ICS-01*, June 2001.

[24] G. Kucuk, K. Ghose, D. Ponomarev, and P. Kogge, “Energy-Efficient Instruction Dispatch Buffer Design for Superscalar Processors”, *ISLPED ‘01*, August 2001.

Appendix A. Circuit Analysis Methodology

Our timing analyses are based on Palacharla's original wakeup and select circuit designs [12,13]. We ported Palacharla's physical design to Taiwan Semiconductor Corporation's (TSMC) 1.8V 0.18µm fabrication technology, using a physical design flow consisting of Cadence and Synopsis design tools. We first optimized Palacharla's original design using Synopsys's AMPS circuit optimization tool (version 5.5). AMPS attempts to optimize circuit latency, power, or area under a given set of constraints. We configured AMPS to optimize circuit latency, with the constraint that transistor area could not increase. AMPS provided the most benefit for the select circuit design, producing a re-sized design that is more than 25% faster, and with only 90% of the original area. AMPS improved wakeup latency nearly 5% with no reduction in area.

Once transistors were sized, timing analysis was performed on a SPICE representation of Palacharla's optimized scheduler design, augmented with parasitic wire delays. Wire parasitics were computed in the same fashion as Palacharla's earlier study, except wire resistance and capacitance was adjusted for the TSMC process. Finally, timing and power analysis was performed using Avant!'s HSPICE circuit tool (version 2001.2), using level 49 typical transistor parameters supplied by Taiwan Semiconductor Corporation for their TSMC 0.18µm 1.8V fabrication process. These parameters are available from MOSIS's secure website [16].

Palacharla's original analyses predate the existence of a functional 0.18µm fabrication technology. Because of this, the device parameters in that work were extrapolated from a Digital Equipment Corporation 0.8µm technology. The timing and power figures for our work were the result of porting Palacharla's original design to TSMC's 0.18µm production fabrication technology and performing timing optimizations using commercial tools configured for the implementation technology. Overall, the ported design is about 24% faster in the commercial technology. The primary factors leading to the faster design are roughly split between faster transistor speed (due to a lower threshold voltage and gate capacitance) and improved logic performance due to better transistor sizing.

Table 3 lists the circuit delay, power, and energy consumption for all analyzed scheduler configurations. For the table, all configurations assume a 4-wide machine capable of producing four results per cycle. In addition, the column labeled $f_{tagload}$ lists the relative tag broadcast bus capacitive load, compared to the same-sized two-tag baseline design. This value indicates the relative decrease in comparator diffusion capacitance, and the relative reduction in tag bus wire length due to elimination of 0-tag reservation stations and denser layout provided by the smaller 1-tag reservation stations.

Table 3: Characteristics of Studied Scheduler Configurations. Scheduler configuration are listed using the notation "x/y/z", where "x" represents the number of 2-tag reservation stations, "y" the number of 1-tag stations, and "z" the number of 0-tag stations. All schedulers are designed for use in a 4-wide microarchitecture and thus have 4 result buses.

Configuration	Total Delay (ps) (wakeup + select)	Total Power (W)	Total Energy (nJ)	$f_{tagload}$
64/0/0	466 (302 + 164)	1.550	0.468	1.0000
20/32/12	383 (219 + 164)	1.435	0.314	0.5625
16/32/16	374 (210 + 164)	1.375	0.289	0.5000
12/32/20	363 (199 + 164)	1.322	0.263	0.4375
8/32/24	355 (191 + 164)	1.250	0.239	0.3750
0/48/16	329 (165 + 164)	1.548	0.255	0.3750
0/40/24	321 (157 + 164)	1.416	0.222	0.3125
0/36/28	315 (151 + 164)	1.372	0.207	0.2813
0/32/32	313 (149 + 164)	1.281	0.191	0.2500
128/0/0	775 (573 + 202)	1.921	1.101	1.0000
40/64/24	552 (350 + 202)	2.064	0.722	0.5625
0/96/32	430 (228 + 202)	2.413	0.550	0.3750
32/0/0	349 (198 + 151)	1.068	0.211	1.0000
10/16/6	317 (166 + 151)	0.938	0.156	0.5625
0/24/8	290 (139 + 151)	0.968	0.135	0.3750